Dynamic Data Race Detection for OpenMP Programs

Yizi Gu   John Mellor-Crummey

Department of Computer Science
Rice University

15 November 2018
Data Race

- A bug in a concurrent program
  - two **concurrent** accesses to the **same memory location**
  - at least one of the memory accesses is a **write**

- **No constraint** on interleaving of instructions

- **Unpredictable** program behavior
OpenMP

- Shared memory parallel programming model
- Widely used node-level programming model on today’s supercomputers
  - typically used to accelerate calculations within an MPI rank
- Supports multiple styles of parallelization, e.g.
  - threads (parallel regions)
  - tasks (implicit and explicit)
  - work sharing (parallel loops, …)
- Multiple synchronization constructs
  - barriers
  - reductions
  - task dependences
  - task wait
  - locks and critical sections
Motivation

• Data races are **hard to debug manually**
  — attaching a debugger or adding print statements
  **biases instruction interleavings**

• **Automatic** data race detection tools are desired

• **Existing tools are not accurate, e.g.**
  — false negatives for loops that carry a dependence
  — false positives for ordered critical sections
Related Work

• Dynamic data race detection
  — cilkscreen for Cilk [Feng & Leiserson SPAA ’97; Cheng et al. SPAA’98]
  — hybrid data race detection algorithm [O’Callahan et al. PPoPP’03]
  — techniques for happens-before analysis
    – reachability queries in DAG [Agrawal et al. SIAM’18]
    – exploit synchronization properties to be efficient, e.g.
      2D-DAGs [Xu et al. PPoPP’18]

• Data race detection for OpenMP programs
  — static analysis
    – Polyhedral analysis [Ye et al. Correctness’18@SC]
  — tools
    – Archer [Atzeni et al. IPDPS’16]; Sword [Atzeni et al. IPDPS’18]
  — benchmark suite
    – DataRaceBench [Liao et al. SC’17]
Contributions

- ROMP - a per-input accurate data race detector for abelian OpenMP programs atop the OMPT first-party tools API
  — https://github.com/zygyz/romp

- Extensions to OpenMP’s OMPT API
  — additional callbacks needed by a precise race detector

- An experimental evaluation of our tool
Approach

• Assign labels to OpenMP tasks to reason about orderings

• Reason about concurrency between tasks
  — multiple concurrent tasks could be executed by the same physical thread in serial order
  — logical concurrency exists regardless of thread schedule

• Maintain access histories for shared variables
  — check if a load or store is a data race with prior accesses
  — record information as needed
1 #pragma omp parallel
2 #pragma omp single
3 #pragma omp parallel for
4 for (int i = 0; i < 4; i ++) {
5     if (i == 0) {
6         #pragma omp task
7         {
8             #pragma omp task
9             {
10            }
11         #pragma omp taskwait
12        }
13    }
14 }
OpenMP Task Labeling

• An OpenMP task label consists of $k$ label segments
  — create new task label upon task creation
  — $k =$ nesting level of OpenMP tasks

• Update task labels at OpenMP synchronization points
  — rely on OMPT callback APIs
  — update fields in task labels to reflect synchronization operations

• Reason about orderings by comparing task labels
  — compare label segments in two task labels
  — $O(k)$ time complexity
<table>
<thead>
<tr>
<th>Field Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>offset</td>
<td>relative id of the worker thread in the team</td>
</tr>
<tr>
<td>span</td>
<td>total number of worker threads in the team</td>
</tr>
<tr>
<td>iteration id</td>
<td>relative id of the iteration in the work-share loop (if applicable)</td>
</tr>
<tr>
<td>taskwait count</td>
<td>number of taskwait encountered by current task</td>
</tr>
<tr>
<td>taskcreate count</td>
<td>number of explicit tasks created by current task</td>
</tr>
<tr>
<td>loop count</td>
<td>number of work-share loops finished by current task</td>
</tr>
<tr>
<td>phase</td>
<td>number of times current task entering/exiting ordered critical</td>
</tr>
<tr>
<td>task waited</td>
<td>true if the task is waited by parent</td>
</tr>
<tr>
<td>task group info</td>
<td>encode task group order info</td>
</tr>
<tr>
<td>segment type</td>
<td>implicit/explicit/logical</td>
</tr>
</tbody>
</table>
Label Updating Rules — I

• creating implicit tasks

  Input  : Parent task label: \( L_p \); Local Thread ID: o; Team Size: s
  Output: A task label L for the newly created implicit task T

  1 \( S \leftarrow \text{CreateNewLabelSegment()} \)
  2 \( S\.segment\_type \leftarrow \text{implicit} \)
  3 \( S\.offset \leftarrow o \)
  4 \( S\.span \leftarrow s \)
  5 \( L \leftarrow L_p\.\text{copy()} \)
  6 \( L\.append(S) \)
  7 \( \text{return } L \)

• creating explicit tasks

  Input  : Parent task label: \( L_p \);
  Output: A task label L for the newly created explicit task T

  1 \( S \leftarrow \text{CreateNewLabelSegment()} \)
  2 \( S\.segment\_type \leftarrow \text{explicit} \)
  3 \( S\.offset \leftarrow 0 \)
  4 \( S\.span \leftarrow 1 \)
  5 \( L \leftarrow L_p\.\text{copy()} \)
  6 \( L\.append(S) \)
  7 \( L_p\.\text{last\_segment\.task\_create\_cnt} += 1 \)
  8 \( \text{return } L \)
Label Updating Rules — II

• **creating logical tasks**

  **Input** : Parent task label: $L_p$; Iteration ID: $i$
  **Output** : New task label $L$

  1. $L ← \text{DiscardLastSegment}(L_p)$
  2. $S ← \text{CreateNewLabelSegment}()$
  3. $S.\text{segment}_\text{type} ← \text{logical}$
  4. $S.\text{offset} ← 0$
  5. $S.\text{span} ← 1$
  6. $S.\text{iteration}_\text{id} ← i$
  7. $L.\text{append}(S)$
  8. return $L$

• **encountering barriers**

  **Input** : Task label $L$ of task encountering the barrier directive
  **Output** : Modified task label $L'$

  1. $S ← L.\text{last}_\text{segment}$
  2. $L' ← \text{DiscardLastSegment}(L)$
  3. $L'.\text{last}_\text{segment}.\text{offset} ← L'.\text{last}_\text{segment}.\text{offset} + L'.\text{last}_\text{segment}.\text{span}$
  4. $L'.\text{append}(S)$
  5. return $L'$
Label Updating Rules — III

- **encountering taskwait**

  **Input**: Task label $L$ of task encountering the `taskwait` directive
  **Output**: Modified task label $L'$

  1. $L' \leftarrow L.copy()$
  2. $L'.last\_segment.taskwait\_cnt \leftarrow L'.last\_segment.taskwait\_cnt + 1$
  3. NotifyExplicitChildrenTaskWait();
  4. return $L'$

- **end of work-share loop**

  **Input**: Task label $L$ of task exiting work-sharing loop construct
  **Output**: Modified task label $L'$

  1. $L' \leftarrow \text{DiscardLastSegment}(L)$
  2. $L'.last\_segment.loop\_cnt \leftarrow L'.last\_segment.loop\_cnt + 1$
  3. return $L'$
Task Labeling: Example

- [offset, span, iter-id, tw, tc, lp, phs, twtd, tg, type]
  - I0: [0,1, {0}, imp]
  - I1: I0[0,2, {0}, imp]
  - I2: I1[0,2, {0}, imp]
  - I3: I1[0,2, {0}, imp]
  - I4: I1[1,2, {0}, imp]
  - L1: I3 [0,4, {0}, lgc]
  - L1': I3 [0,4, {tc:1}, lgc]
  - L1: I3 [0,4, {0}, lgc]
  - L1': I3 [0,4, {tc:1}, lgc]
  - E1: L1[0,1, {0}, exp]
  - E1': L1[0,1, {tc:1}, exp]
  - E1''': L1[0,1, {tc:1, tw:1}, exp]
  - E1: L1[0,1, {0}, exp]
  - E1': L1[0,1, {tc:1}, exp]
  - E1'': L1[0,1, {tc:1, tw:1}, exp]
  - I4 after S2: I1[1,2, {lp:1}, imp]
  - I4 after S4: I0[2,2, {0}, imp][1,2, {lp:1}, imp]
Benefits of OpenMP Task Labeling

- Decentralized reachability query
  - conduct query by comparing two labels
  - multiple queries can execute in parallel

- Concisely represents logical concurrency in OpenMP constructs
  - iterations in a work-sharing loop construct as logically concurrent tasks
  - synchronization encoded in task labels
Per-Task Metadata

• Problem: some synchronization is unsuited for task labels

• Approach
  — use OMPT callbacks to notify ROMP of synchronization
  — store sync info in data structures associated with each task

• Examples
  — explicit task dependences
    – task dependence graph built according to dependence variables
    – search directed path in the graph for ordering relation
    – only inspected if no order can be determined using task labels
  — reductions
    – monitor entry/exit of reduction region by OMPT callback
  — critical/atomic sections
    – maintain a set of locks held during each memory access
A Hybrid Data Race Detection Algorithm

- Invoke this algorithm to check if an access races with others

- When to prune accesses?
  - current access ordered with a prior access
  - one access performed with fewer locks
    - discard an access record in the history
    - don’t add a record for the current access

- Adapts All-Sets algorithm for Cilk to enable parallel execution [Cheng et al. SPAA’98]

```plaintext
Input: Memory address being accessed: l
      Current access record <ε,a,h>
      History of accesses: History
Output: Report data race on l if current access is racing with history accesses

1   skip_current ← FALSE
2  foreach <ε’, a’, h’> ∈ History[l] do
3    if ε || ε’ ∩ h ∩ h’ = ∅ ∧ (a = w ∨ a’ = w) then
4      report a data race
5    if ((a’ = w ∧ a = w) ∨ a’ = r) ∧ h’ ⊇ h ∧ ε’ → ε then
7      continue
8
9    if ((a’ = r ∧ a = r) ∨ a’ = w) ∧ h ⊇ h’ ∧ ε’ → ε then
10   skip_current ← TRUE
11
12 if skip_current is FALSE then
13   History[l] ← History[l] ∪ <ε, a, h>
```
Data Environment Tracking: Motivation

- A task may use storage on the stack and in the heap
- When a task finishes, it releases its storage
- Later a logically concurrent task could reuse that storage
- At run time we must track such information so that we don’t report this as a data race
Data Environment Tracking: Method

- Maintain shared/private state for each memory cell
- Mark a memory cell as private if
  - its address falls within range of stack of accessing thread
  - its address is below the accessing task’s base
- Mark a memory cell as shared if
  - its address falls out of range of stack of accessing thread
  - accessed by exp. tasks, its addr. falls out of task private region
- Mark memory cells as deallocated upon task schedule point
  - implicit tasks are bound to physical threads, private variable is deallocated when finishes
  - for explicit tasks, mark as deallocated upon task schedule
Correctness

- Abelian property of a parallel program [Cheng et al. SPAA’98]
  - a program is abelian if
    - any critical sections protected by the same lock commute
    - thread schedule does not affect control flow
    - all locks are properly nested

- Provided that a program is abelian and does not use SIMD construct
  - if data races exist with respect to a shared variable, ROMP reports at least one
    - no need to report all instances of the same data race
  - if no data race exists with respect to a shared variable, ROMP does not report one
Extensions to OMPT APIs

• ompt_callback_reduction
  — notify tools of entering/exiting reduction region

• ompt_callback_dispatch
  — notify tools of dispatching of a logical task in a work-sharing construct

• ompt_get_task_memory
  — get the range of memory allocated for a task
Shadow Memory

• Store access histories

• Two level page table
  — similar to Valgrind [N.Nethercote et al. VEE’07]

• Use locks to protect concurrent accesses to shadow memory
  — when checking multiple concurrent access to the same location
  — mutual exclusion to avoid data races when maintaining shadow memory
Optimization

• Don’t check accesses to read-only data

• Dynamic checking shortcut
  — lock contention when accessing shadow memory
    – both checking procedures run in parallel
    – both check accesses to the same memory location
    – at least one of the checks is for a write access
  — directly report the data race
    – no need to invoke the checking protocol
**Implementation**

- Implement ROMP functionality as a library

- Use a binary instrumentation tool to insert a call to ROMP to check if a load or store is involved in a data race
  — DynInst [CS@UW-Madison]

- Maintain access histories in shadow memory associated with each program variable
Evaluation Metrics

- **Recall:** \( \frac{TP}{TP + FN} \)
  - fraction of real races reported

- **Precision:** \( \frac{TP}{TP + FP} \)
  - fraction of races reported that are real

- **Accuracy:** \( \frac{TP + TN}{TP + TN + FP + FN} \)
  - correct check results / total checks
Evaluation

• System: single-node, four 12-core AMD Opterons, 128GB of memory

• Evaluate accuracy
  — compare ROMP with Archer using DataRaceBench
  — run test script provided by DataRaceBench
    – modified test script to add option for testing ROMP
  — not including test cases for containing SIMD constructs
    – limitation of dynamic data race detectors

• Evaluate performance
  — compare ROMP with Archer using OmpSCR
Results: Accuracy

ROMP is more accurate than the best prior race detector for OpenMP programs

<table>
<thead>
<tr>
<th></th>
<th>Archer</th>
<th>ROMP</th>
</tr>
</thead>
<tbody>
<tr>
<td>Precision</td>
<td>0.79</td>
<td>1.0</td>
</tr>
<tr>
<td>Recall</td>
<td>0.79</td>
<td>1.0</td>
</tr>
<tr>
<td>Accuracy</td>
<td>0.78</td>
<td>1.0</td>
</tr>
</tbody>
</table>
## Results: Performance

<table>
<thead>
<tr>
<th></th>
<th>Run time Overhead</th>
<th>Memory Overhead</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Archer</td>
<td>ROMP</td>
</tr>
<tr>
<td>Mean</td>
<td>130</td>
<td>91.5</td>
</tr>
<tr>
<td>Median</td>
<td>8.70</td>
<td>8.23</td>
</tr>
<tr>
<td>Geo-Mean</td>
<td>18.6</td>
<td>20.5</td>
</tr>
</tbody>
</table>

- Run time overhead
  - Archer is 10% faster on OmpSCR

- Space overhead
  - ROMP uses 2.5x less space on OmpSCR
Results: Scalability

A parallel quick sort program
Input size: 1 million

<table>
<thead>
<tr>
<th>Input size</th>
<th>Time overhead</th>
</tr>
</thead>
<tbody>
<tr>
<td>1k</td>
<td>2.7x</td>
</tr>
<tr>
<td>2k</td>
<td>2.1x</td>
</tr>
<tr>
<td>4k</td>
<td>7.7x</td>
</tr>
<tr>
<td>8k</td>
<td>5.9x</td>
</tr>
<tr>
<td>16k</td>
<td>11x</td>
</tr>
<tr>
<td>32k</td>
<td>21x</td>
</tr>
<tr>
<td>64k</td>
<td>27x</td>
</tr>
</tbody>
</table>

A parallel quick sort program
Number of threads: 16
Summary and Ongoing Work

• Contributions
  — ROMP: a per-input accurate race detector for abelian OpenMP programs
    – more precise than prior OpenMP race detectors
  — experimental evaluation
    – ROMP uses comparable time and less space than Archer
  — extensions to OpenMP’s OMPT API for precise race detection

• Ongoing work
  — instrument all dynamic libraries
  — reduce overhead and improve scalability
    – prune long access histories for widely-shared data
      current pruning criteria are correct but conservative
      exploit properties of OpenMP parallel constructs to prune more access records: keep only necessary rather than sufficient records